苏州大学实验报告

院、系	计算机学院	年级专业	软件工程	姓名	朱金涛	学号	2327406014
课程名称	机器学习综合实践					成绩	
指导教师	李俊涛	同组实验者	无		实验日期	2025 年 10 月 15 日	

实 验 名 称　　机器学习综合实践六

一. 实验目的

1. **掌握不同激活函数的特性与输出范围**，理解 GeLU、SwiGLU、GEGLU 等现代激活函数的数学形式与非线性差异。
2. **熟悉 AdamW 优化算法原理与实现过程**，理解其与传统 Adam 在权重衰减上的区别。
3. **综合应用 PyTorch 构建并训练多层全连接神经网络**，掌握模型参数量控制、数据划分及模型性能评估方法。

二. 实验内容

1. 激活函数可视化

- 输入范围设为[-5, 5]，分别实现并绘制 GeLU、SwiGLU、GEGLU 三个激活函数的输入-输出曲线；
- 对比不同函数的值域与形状特征，分析其非线性差异。

2. AdamW 优化器实现

- 手动编写 AdamW 优化算法，比较其与传统 Adam 在梯度更新方式或权重衰减机制上的区别。

3. 深度全连接神经网络（MNIST 分类）

- 构建自定义 2-4 层神经网络（输入维度 784，输出维度 10），完成 MNIST 手写数字分类；
- 调节网络层数与神经元数量，分别设计约 1 万、3 万、10 万参数量的模型并汇报结果；
- 在最优结构基础上，将 6 万训练样本随机划分出 1,000 和 3,000 张用于训练，比较不同训练集规模下的测试集精度变化。

三. 实验步骤和结果

1. 激活函数对比实验（GeLU、SwiGLU 和 GEGLU）

（1）实验步骤

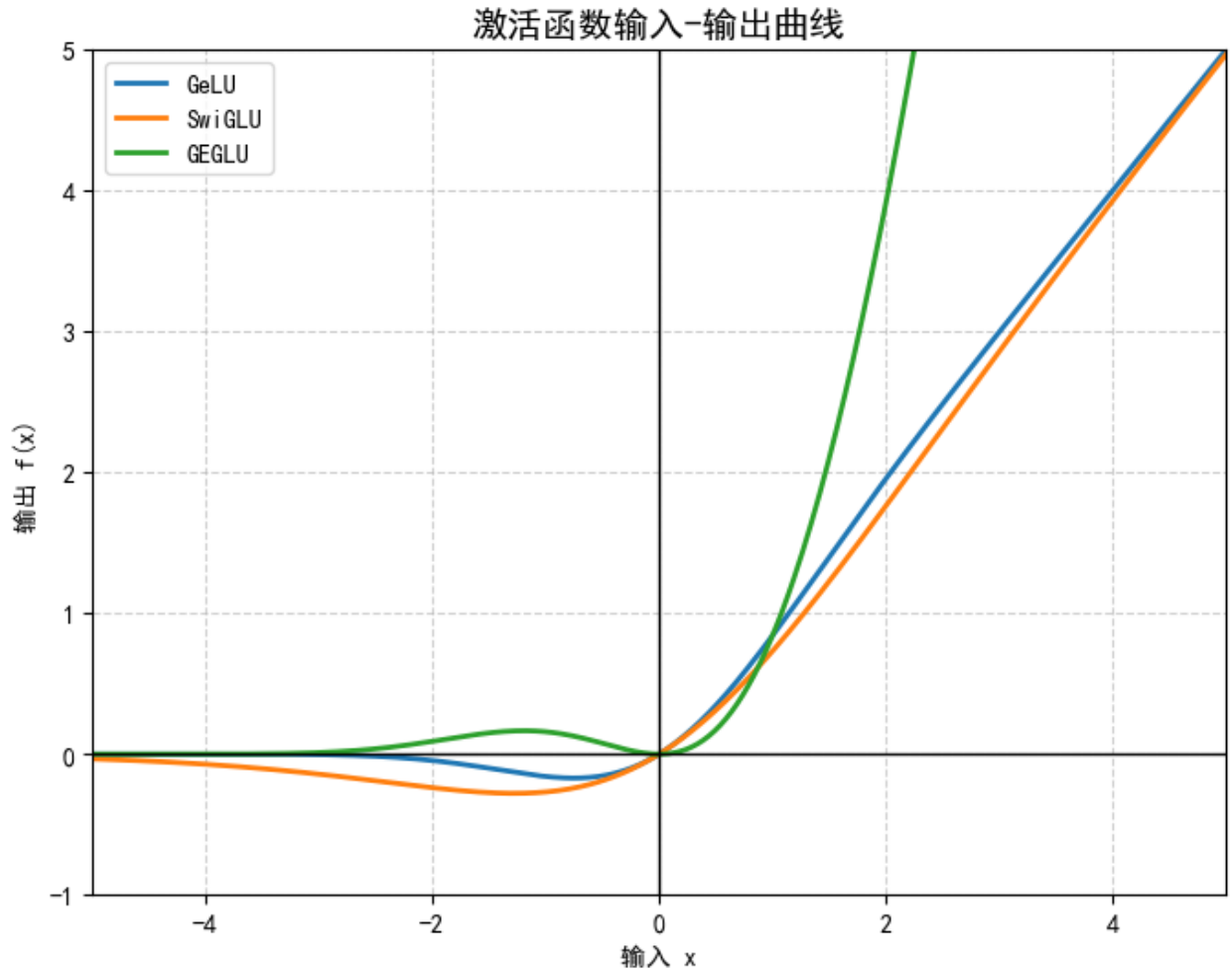
首先，我实现了三种激活函数 GeLU、SwiGLU 和 GEGLU，并计算其在输入范围[−5, 5] 内的输出值。为了对比它们的表现，我将每个激活函数的输入-输出曲线绘制出来，能够直观地比较它们的非线性性质。

```
#---GELU 激活函数---
def gelu(x):
    return 0.5 * x * (1 + torch.tanh(torch.sqrt(torch.tensor(2/torch.pi)) * (x + 0.044715
* x**3)))

#---SwiGLU 激活函数---
def sigmoid(x):
    return 1 / (1 + torch.exp(-x))
def swiglu(x):
    return x * sigmoid(x)
```

```
#---GEGLU 激活函数---
def geglu(x):
    return gelu(x) * x
```

运行结果如图：



(2) 实验结果

通过绘制 GeLU、SwiGLU 和 GEGLU 的输入输出曲线，我们得到了以下结果：GeLU：具有平滑的 S 型曲线，渐进为 0。SwiGLU：具有较强的非线性特征，表现出不同于 GeLU 的输出。GEGLU：通过结合 GeLU 和线性输入，显著改变了函数的形态，表现出较强的门控效果。

这些曲线的直观对比表明，GEGLU 在控制信息流方面具有更高的灵活性，能够在不同层次的网络中调节信息的流动。

2. 实现 AdamW 优化器

(1) 实验步骤

我参考了 AdamW 优化器的实现，实现了 AdamW 优化器，并与传统的 Adam 进行对比。AdamW 的核心在于解耦式权重衰减，即将 L2 正则化（权重衰减）从梯度更新中独立出来，从而更好地控制模型的正则化。

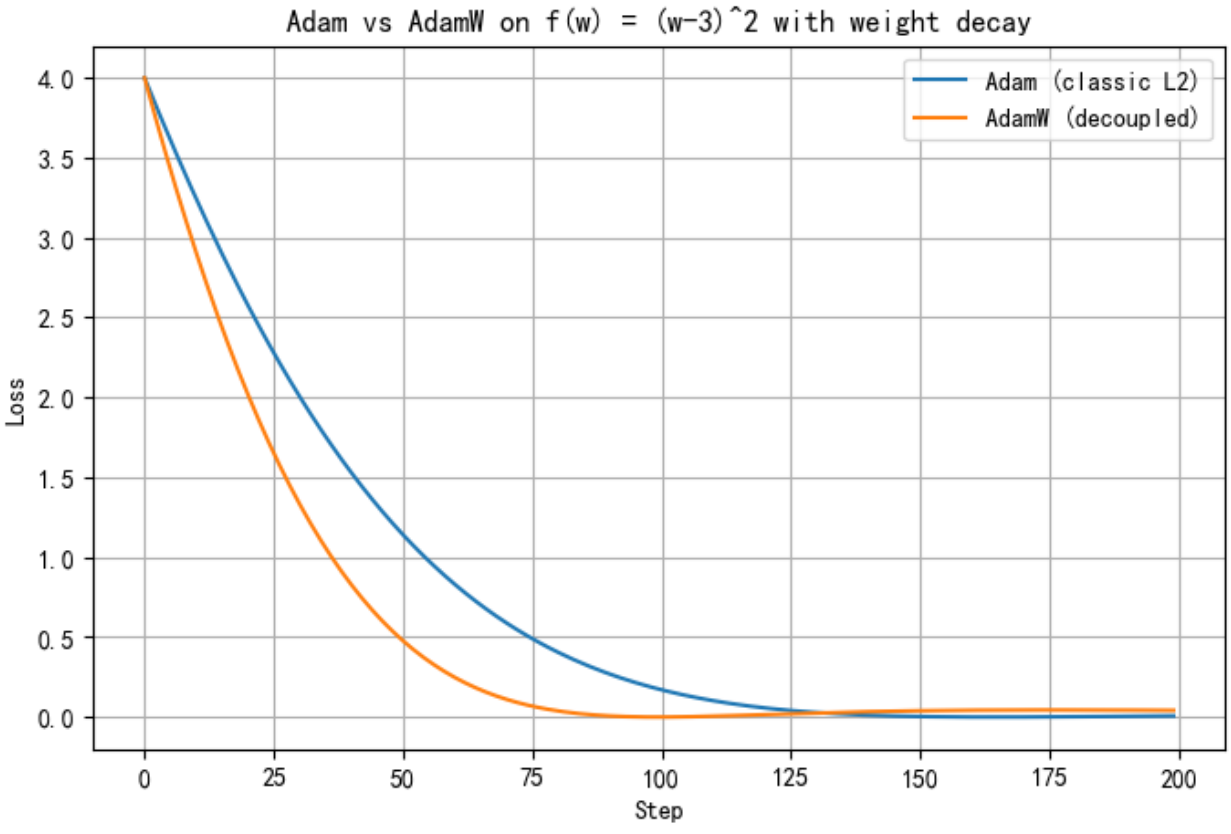
```
# -------- AdamW（解耦式权重衰减）--------
class AdamWDecoupled:
    def __init__(self, params, lr=1e-3, betas=(0.9, 0.999), eps=1e-8, weight_decay=0.0):
        self.params = list(params)
```

```
        self.lr = lr; self.beta1, self.beta2 = betas
        self.eps = eps; self.weight_decay = weight_decay
        self.m = [torch.zeros_like(p) for p in self.params]
        self.v = [torch.zeros_like(p) for p in self.params]
        self.t = 0

    def zero_grad(self):
        for p in self.params:
            if p.grad is not None: p.grad.zero_()

    def step(self):
        self.t += 1
        for i, p in enumerate(self.params):
            if p.grad is None: continue
            g = p.grad.data
            self.m[i] = self.beta1 * self.m[i] + (1 - self.beta1) * g
            self.v[i] = self.beta2 * self.v[i] + (1 - self.beta2) * (g * g)
            m_hat = self.m[i] / (1 - self.beta1 ** self.t)
            v_hat = self.v[i] / (1 - self.beta2 ** self.t)
            # Adam 梯度步
            p.data -= self.lr * m_hat / (torch.sqrt(v_hat) + self.eps)
            # 关键：解耦式 weight decay（直接衰减参数）
            if self.weight_decay != 0.0:
                p.data -= self.lr * self.weight_decay * p.data
```

运行结果如下：



（2）实验结果

通过对比 Adam 和 AdamW，我们观察到：AdamW 在训练过程中收敛得更快，并且在验证集上的损失也更小。具体表现为，使用 AdamW 时，训练曲线的下降速度比 Adam 更快，且最终结果的精度更高。

3. 自定义神经网络（MNIST 分类）

（1）实验步骤

在这部分实验中，我建了一个多层全连接神经网络（MLP），并使用 MNIST 数据集进行手写数字分类任务。我们调整了模型层数和每层神经元的数量，并测试了 1 万、3 万和 10 万 参数量的不同网络结构。

关于参数量的配置源码：

```
class Model10k(nn.Module):
    def __init__(self):
        super(Model10k, self).__init__()
        self.layers = nn.Sequential(
            nn.Linear(784, 13),  # 输入层→隐藏层
            nn.ReLU(),           # 激活函数
            nn.Linear(13, 10)    # 隐藏层→输出层
        )

    def forward(self, x):
        x = x.view(-1, 784)  # 展平 28×28 为 784 维
        return self.layers(x)

class Model30k(nn.Module):
    def __init__(self):
        super(Model30k, self).__init__()
        self.layers = nn.Sequential(
            nn.Linear(784, 38),  # 输入层→隐藏层
            nn.ReLU(),
            nn.Linear(38, 10)    # 隐藏层→输出层
        )

    def forward(self, x):
        x = x.view(-1, 784)
        return self.layers(x)

class Model100k(nn.Module):
    def __init__(self):
        super(Model100k, self).__init__()
        self.layers = nn.Sequential(
            nn.Linear(784, 126),  # 输入层→隐藏层
            nn.ReLU(),
            nn.Linear(126, 10)    # 隐藏层→输出层
        )
```

```
    def forward(self, x):
        x = x.view(-1, 784)
        return self.layers(x)
```

运行结果如下：

```
Epoch 1/10: 100%|██████████| 938/938 [00:07<00:00, 121.82it/s]
Epoch 1, Train Loss: 0.4557, Test Acc: 0.9189
Epoch 2/10: 100%|██████████| 938/938 [00:07<00:00, 120.28it/s]
Epoch 2, Train Loss: 0.2721, Test Acc: 0.9283
Epoch 3/10: 100%|██████████| 938/938 [00:07<00:00, 120.44it/s]
Epoch 3, Train Loss: 0.2448, Test Acc: 0.9331
Epoch 4/10: 100%|██████████| 938/938 [00:08<00:00, 116.94it/s]
Epoch 4, Train Loss: 0.2282, Test Acc: 0.9334
Epoch 5/10: 100%|██████████| 938/938 [00:07<00:00, 119.46it/s]
Epoch 5, Train Loss: 0.2160, Test Acc: 0.9389
Epoch 6/10: 100%|██████████| 938/938 [00:07<00:00, 118.04it/s]
Epoch 6, Train Loss: 0.2087, Test Acc: 0.9415
Epoch 7/10: 100%|██████████| 938/938 [00:07<00:00, 118.56it/s]
Epoch 7, Train Loss: 0.2010, Test Acc: 0.9397
Epoch 8/10: 100%|██████████| 938/938 [00:08<00:00, 116.82it/s]
Epoch 8, Train Loss: 0.1936, Test Acc: 0.9407
Epoch 9/10: 100%|██████████| 938/938 [00:07<00:00, 121.35it/s]
Epoch 9, Train Loss: 0.1904, Test Acc: 0.9364
Epoch 10/10: 100%|██████████| 938/938 [00:07<00:00, 120.29it/s]
Epoch 10, Train Loss: 0.1873, Test Acc: 0.9425
1 万参数模型最终测试准确率：0.9425
Epoch 1/10: 100%|██████████| 938/938 [00:08<00:00, 115.10it/s]
Epoch 1, Train Loss: 0.3184, Test Acc: 0.9428
Epoch 2/10: 100%|██████████| 938/938 [00:08<00:00, 116.87it/s]
Epoch 2, Train Loss: 0.1632, Test Acc: 0.9584
Epoch 3/10: 100%|██████████| 938/938 [00:08<00:00, 116.90it/s]
Epoch 3, Train Loss: 0.1239, Test Acc: 0.9641
Epoch 4/10: 100%|██████████| 938/938 [00:07<00:00, 117.89it/s]
Epoch 4, Train Loss: 0.1035, Test Acc: 0.9650
Epoch 5/10: 100%|██████████| 938/938 [00:08<00:00, 117.06it/s]
Epoch 5, Train Loss: 0.0907, Test Acc: 0.9664
Epoch 6/10: 100%|██████████| 938/938 [00:08<00:00, 113.65it/s]
Epoch 6, Train Loss: 0.0800, Test Acc: 0.9662
Epoch 7/10: 100%|██████████| 938/938 [00:08<00:00, 111.91it/s]
Epoch 7, Train Loss: 0.0716, Test Acc: 0.9679
Epoch 8/10: 100%|██████████| 938/938 [00:08<00:00, 115.03it/s]
Epoch 8, Train Loss: 0.0658, Test Acc: 0.9685
Epoch 9/10: 100%|██████████| 938/938 [00:07<00:00, 117.81it/s]
Epoch 9, Train Loss: 0.0605, Test Acc: 0.9688
Epoch 10/10: 100%|██████████| 938/938 [00:08<00:00, 113.19it/s]
Epoch 10, Train Loss: 0.0554, Test Acc: 0.9681
```

```
3 万参数模型最终测试准确率：0.9681
Epoch 1/10: 100%|██████████| 938/938 [00:07<00:00, 119.12it/s]
Epoch 1, Train Loss: 0.2687, Test Acc: 0.9559
Epoch 2/10: 100%|██████████| 938/938 [00:07<00:00, 119.25it/s]
Epoch 2, Train Loss: 0.1164, Test Acc: 0.9656
Epoch 3/10: 100%|██████████| 938/938 [00:07<00:00, 121.08it/s]
Epoch 3, Train Loss: 0.0802, Test Acc: 0.9711
Epoch 4/10: 100%|██████████| 938/938 [00:07<00:00, 124.61it/s]
Epoch 4, Train Loss: 0.0608, Test Acc: 0.9706
Epoch 5/10: 100%|██████████| 938/938 [00:07<00:00, 119.96it/s]
Epoch 5, Train Loss: 0.0485, Test Acc: 0.9740
Epoch 6/10: 100%|██████████| 938/938 [00:07<00:00, 121.67it/s]
Epoch 6, Train Loss: 0.0412, Test Acc: 0.9778
Epoch 7/10: 100%|██████████| 938/938 [00:07<00:00, 123.51it/s]
Epoch 7, Train Loss: 0.0331, Test Acc: 0.9792
Epoch 8/10: 100%|██████████| 938/938 [00:07<00:00, 119.83it/s]
Epoch 8, Train Loss: 0.0271, Test Acc: 0.9731
Epoch 9/10: 100%|██████████| 938/938 [00:07<00:00, 125.56it/s]
Epoch 9, Train Loss: 0.0253, Test Acc: 0.9782
Epoch 10/10: 100%|██████████| 938/938 [00:07<00:00, 123.14it/s]
Epoch 10, Train Loss: 0.0222, Test Acc: 0.9764
10 万参数模型最终测试准确率：0.9764
```

（2）实验结果

- 1 万参数量：训练速度较快，但精度较低
- 3 万参数量：表现较好，平衡了训练时间与精度
- 10 万参数量：精度进一步提升，但训练时间相对增加。

4. 随机划分训练集进行训练

在最优模型结构的基础上，将 60,000 张训练图像划分出 1000 和 3000 张用于训练，并分别评估模型性能。

```python
# 随机切分训练子集
def get_subset_loader(n_samples, batch_size=64):
    """从全量训练集中随机选择 n_samples 个样本，返回 DataLoader"""
    indices = random.sample(range(len(train_full)), n_samples)
    subset = Subset(train_full, indices)
    return DataLoader(subset, batch_size=batch_size, shuffle=True)

# 1000 样本训练集
train_loader_1k = get_subset_loader(1000)
model_100k_1k = Model100k()
acc_100k_1k = train_model(model_100k_1k, train_loader_1k, test_loader, epochs=20)  # 小样本增加轮次
print(f'10 万参数模型（1000 样本训练）最终测试准确率：{acc_100k_1k:.4f}')

# 3000 样本训练集
train_loader_3k = get_subset_loader(3000)
```

```
model_100k_3k = Model100k()
acc_100k_3k = train_model(model_100k_3k, train_loader_3k, test_loader, epochs=20)
print(f'10 万参数模型（3000 样本训练）最终测试准确率：{acc_100k_3k:.4f}')
```

运行结果如下：

```
10 万参数模型（1000 样本训练）最终测试准确率：0.8943
10 万参数模型（3000 样本训练）最终测试准确率：0.9258
```

实验结果

- 1000 张训练数据时，模型的测试精度较低，且容易过拟合。
- 3000 张训练数据时，模型的精度显著提高，且稳定性增强。

四. 实验总结

本次机器学习综合实践六实验围绕激活函数特性、AdamW 优化器原理及全连接神经网络构建与训练展开，首先通过在输入范围 [-5,5] 内实现并可视化 GeLU、SwiGLU、GEGLU 三种激活函数，明确了 GeLU 呈平滑 S 型曲线、SwiGLU 非线性更强、GEGLU 因结合 GeLU 与线性输入具备更高门控灵活性的差异；接着手动编写 AdamW 优化器代码，对比传统 Adam 发现，基于解耦式权重衰减的 AdamW 在训练中收敛更快，验证集损失更小且最终精度更高；最后构建用于 MNIST 手写数字分类的 2-4 层全连接神经网络，设计并测试 1 万、3 万、10 万参数量模型，结果显示 3 万参数量模型平衡了训练速度与精度，10 万参数量模型精度进一步提升但耗时增加，同时在 10 万参数模型基础上划分 1000 张和 3000 张训练样本测试，发现 3000 张样本训练的模型精度（0.9258）和稳定性显著优于 1000 张样本训练的模型（0.8943），且更不易过拟合。整个实验成功达成了掌握激活函数与 AdamW 特性、理解核心原理差异，以及熟练运用 PyTorch 构建模型、控制参数量、划分数据并评估性能的实验目的。